2

NPSCS-91-007

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AN OBJECT-ORIENTED DESIGN METHODOLOGY

Everton G. de Paula, Captain, Brazilian AF
Michael L. Nelson, Major USAF

January 1991

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100

This report was prepared in conjunction with research funded by the Naval Postgraduate School Research Council.

[illegible]

MICHAEL L. NELSON
Assistant Professor
of Computer Science

Reviewed by:

[illegible]

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

P. J. Marto

PAUL J. MARTO
Dean of Research

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is limited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-91-007	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Department of Computer Science Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School; 8b. OFFICE SYMBOL (If applicable) CS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct Funding
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
An Object-Oriented Design Methodology

12. PERSONAL AUTHOR(S)
Everton G. DePaula and Michael L. Nelson

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Summary	FROM ____ TO ____	91 Jan 31	32

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Object-Oriented Design, Object-Oriented Programming, Inheritance, Class Hierarchy, Low Cost Combat Direction System

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

To date, there is no design methodology that is universally accepted by the object-oriented community. Several such methodologies, however, have been proposed. They are all somewhat similar in their approach to identifying and defining the objects and in organizaing them into class hierarchies.

The methodology proposed in this paper is the result of a research project in object-oriented design, and benefitted from the experience acquired during the design of the Tactical Database for the Low Cost Combat Direction System.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael L. Nelson	22b. TELEPHONE (Include Area Code) (408)646-2449; 22c. OFFICE SYMBOL CS NE

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE
S/N 0102-LF-014-6603 UNCLASSIFIED

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Object-oriented programming (OOP) is becoming widely accepted as a viable approach to nearly any programming project. It may be the only feasible strategy for those types of projects that have not been served well by more traditional approaches. Additionally, it is also proving to be quite useful for more conventional projects.

However, the field is currently lacking an established design methodology. The basic ideas of objects, classes, and inheritance are all becoming fairly well established, but there is no set of rules or guidelines available to help determine just what is the "best" approach to the design of an OOP application. What is needed is something along the lines of the normalization technique of relational databases [Ullm82] (for even if a normalized set of relations is not "ideal", it at least provides a "good starting point").

Unfortunately, we are not yet to the point of being able to provide a set of steps which will always result in an ideal design. Rather, what we have developed is a series of questions/considerations which should help the object-oriented designer to arrive at what is at least a reasonably good starting point. Our approach utilizes the knowledge gained from surveying existing literature and the experience gained during the design of the Tactical Database for the Low Cost Combat Direction System (LCCDS)[1] [dePa90, Ross89, SS90].

1.1 OBJECT-ORIENTED TERMINOLOGY

It is assumed that the user is at least somewhat familiar with the basic concepts of OOP. However, a brief introduction is included as the terminology often varies greatly from one system to the next.[2]

The primitive element of an object-oriented programming language is the object. An *object* encapsulates both data (attributes or variables) and behavior (functions, operations, or methods). That is, an object remembers certain information and it knows how to perform certain operations. [WWW90]

Individual objects may have similarities. In any given application, some objects will behave differently from one another while other objects will behave in a like manner. Objects which share both the same behavior and structure are said to belong to the same class. A *class* is a generic specification for an arbitrary number of similar objects. A class can be thought of as a template for a specific kind of object. An object belonging to a class is said to be an *instance* of that class. [WWW90]

The terms class and type are sometimes used interchangeably. Herein the term type will be used in accordance with more conventional programming languages. A *type* is any data type, whether it is provided by the

[1]It should be realized that while this paper uses the LCCDS Tactical Database for examples, some of the objects, classes, and issues involved have been simplified - the purpose of this paper is to present our design methodology, not the LCCDS Tactical Database itself. For more precise information on the LCCDS Tactical Database, refer to [dePa90].

[2]For a more complete introduction, the reader should refer to [Nels90, SB86, Wegn87].

system (e.g., integer, real, character, etc) or user defined (i.e., a class). Additionally, we may occasionally refer to a set of values, such as (0..59) or {N,S}, as a type. Sets of values such as these can easily be defined either as a restriction of a system-supplied type or as a user-defined class.

The attributes that characterize a class (object) are referred to as *variables* and the operations (or procedures) defined for that class (object) are referred to as *methods*. The variables that form a class can be divided into two categories: class variables and instance variables. A *class variable* is shared in both name and value by all instances (objects) of a class. An *instance variable* is shared in name only by all instances (objects) of a class. That is, each object has its own local version of an instance variable, while all the objects of the class access the same class variable. [Nels90]

A *composite object* (also called an *aggregate object*) is simply an object that contains other objects as instance variables or as class variables. This occurs when an instance variable or a class variable is itself an object.

In a composite object, the relationship between the object and its variables is described as an 'is-part-of' or a 'has-a' relationship, in which the object is viewed as a whole and its variables are viewed as its component parts. That is, each component object is-part-of the composite object. Alternatively, it can also be said that the composite object has-a component object.

Inheritance can be formally (but simply) defined as a code sharing mechanism. It allows a new class to be defined based upon the definition of an existing class, without having to copy all of the code of this other class. A class inherits both behavior (i.e., methods) and structure (i.e., variables) from another class, called a *superclass*. A class which inherits from some superclass is called a *subclass*. Inheritance is sometimes referred to as an 'is-a' relationship, in contrast to the 'has-a' or 'is-part-of' relationship of composite objects. [Nels90]

Single inheritance is when a class is allowed to have only one superclass. *Multiple inheritance* allows a class to have several superclasses. *Selective inheritance* (also called *exclusive inheritance*) is a form of inheritance in which only some methods and/or variables are inherited from the superclass(es).

An *abstract class* is one that is not intended to produce instances. Abstract classes are designed for inheritance purposes only (i.e., to provide structure and/or behavior to one or more subclasses). A class that is designed to be instantiated is called a *concrete class*. Concrete classes are designed primarily so that their instances are useful, and secondly so that they may also be usefully inherited from. [WWW90]

Both abstract and concrete classes may make use of inheritance. However, abstract classes are generally used only to factor out attributes and/or operations that are common to more than one class, putting them all together in one place so that any number of subclasses may make use of them. [WWW90]

Figure 1 (an extension of [Nels90]) shows how class definitions will be represented in a language-independent manner in this paper. Note the convention that class names are written in CAPITALS, variable and method names are written in SMALL CAPITALS, variable types are written in normal type, and variable values are written in *italics*. This form of representation will be used throughout this paper.

```
CLASS: CLASS NAME
  Superclasses:            SUPERCLASS 1, SUPERCLASS 2, ...
  Class Variables:         CLASS VARIABLE 1: type [default value]
                           CLASS VARIABLE 2: type [default value]
                                  ⋮
  Instance Variables:      INSTANCE VARIABLE 1: type [default value]
                           INSTANCE VARIABLE 2: type [default value]
                                  ⋮
  Methods:                 METHOD 1
                           METHOD 2
                                  ⋮
```

Figure 1: Language-Independent Class Definition

1.2 OBJECT-ORIENTED DESIGN

The process of object-oriented design begins with the discovery of the classes and objects that represent the problem domain. It stops whenever the designer finds that there are no new abstractions or mechanisms that might be used to modify the class hierarchy [Booc91]. The result of an object-oriented design is a hierarchy of classes [WEK90].

The class hierarchy is based on inheritance. Inheritance is used to create a new class of objects as a refinement of another, to factor out the similarities between classes, and to design and specify only the differences for the new class. In this way, new classes can be created quickly and efficiently. [WWW90]

The main reason for organizing the classes in a hierarchy is that this type of organization allows the user to obtain simpler and more easily maintainable code while saving considerable storage space. It allows the developer to quickly and easily reuse existing code, thereby minimizing code duplication.

1.2.1 DATA-DRIVEN vs RESPONSIBILITY-DRIVEN DESIGN

Object-oriented design has been categorized as being either data-driven or responsibility-driven [WW89]. With a *data-driven* approach, it is the structure of the data which drives the design. Two questions are central to this approach: (1) What (structure) does this object represent?; and (2) What operations can be performed by this object? A data-driven approach is said to be easier for programmers experienced with traditional procedural languages, as they simply adapt their previous experience to the design of the object-oriented system. However, it is claimed that this approach violates encapsulation in that it makes the structure of the object part of its definition, and that this leads to operations which reflect the given structure. [WW89]

The *responsibility-driven* approach, which has preserving encapsulation as its primary goal, is based upon the *client/server* model. This model describes the interaction between the client and the server. A *contract* is established which specifically states the ways in which the client can interact with the server. The supposed advantage of this approach is that it concentrates on what the server does for the client rather than on how the server does it. The responsibility-driven approach focuses on the contract by asking: (1) What actions is this object responsible for?; and (2) What information does this object share? (note that this shared information may or may not be part of the structure of the object; it could, for example, compute the data or request the information from another object). [WW89]

We are not totally convinced of this design dichotomy. We believe that the two approaches can be construed as being essentially one and the same, as long as the designer strives for a high degree of encapsulation. For example, a data-driven approach may start out with an initial structure for an object (class). However, if the design then proceeds from a responsibility-driven point of view, this initial structure only means that the object is responsible for being able to provide that information on demand; it makes no difference whether or not the object actually maintains that information, computes it, or requests it from some other object. This is essentially the approach that we have taken in our work.

2 THE METHODOLOGY

Our methodology consists of the following steps:

(1) Identification of the objects and classes.
   (a) Initial definition of the objects and classes.
   (b) Analysis of the object's variables.
   (c) Analysis of the object's methods.
(2) Refinement of the objects and classes.
   (a) Addition of necessary information.
   (b) Elimination of redundant information.
   (c) Determination of class and instance variables.
   (d) Identification of composite objects.
(3) Organization of the classes into a hierarchy.
   (a) Analysis of the implementation language.
   (b) Construction of the hierarchies.
   (c) Review of the classes' variables/methods.

It is important to note that the design as a whole is an iterative process. That is, each step may need to be reviewed several times before the hierarchy of classes constructed is considered satisfactory.

2.1 IDENTIFICATION OF THE OBJECTS AND CLASSES

The primary motivation for identifying objects and classes is to match the system as closely as possible to the conceptual view of the real world [CY90]. The design of an object-oriented system begins with the identification of the objects, yet this identification is perhaps the most challenging phase of an object-oriented design process [WEK90].

2.1.1 INITIAL DEFINITION OF THE OBJECTS AND CLASSES

The purpose of this step is to come up with a list of potential objects and/or classes. The description of each object in the list contains the object's initial set of variables and methods.

Identification of the objects and classes requires a considerable knowledge of the problem domain (problem space). The developer must study the problem requirements as well as learn the terminology and fundamentals of the problem domain.

It has been suggested that a preliminary set of candidate classes and objects can usually be derived from the following sources [SM88]:

Tangible things: Cars, radar data, sensors, airplanes.

Roles: Mother, engineer, systems analyst.

Events: Landing, request, print.

Interactions: Loan, meeting, intersection.

A similar list of sources of potential classes and objects has also been suggested [Ross87]:

People: Humans who carry out some function.

Places: Areas set aside for people or things.

Things: Physical objects that are tangible.

Organizations: Formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals.

Concepts: Principles or ideas not tangible per se.

Events: Things that happen, usually to something else at a given date and time, or as steps in an ordered sequence.

All of these ideas are useful in finding candidate objects. It should be noted, however, that the sources of potential objects are virtually unlimited. The problem space is, obviously, the best place to begin the search.

2.1.2 ANALYSIS OF THE OBJECT'S VARIABLES

The purpose of this step is to analyze the variables defined for the objects which resulted from step 2.1.1 and list those observations that are considered to be significant to the construction of the class hierarchy. The following guidelines govern this phase:

Analyze the object's variables to make certain that each variable is absolutely necessary for the description of the object (class). Unnecessary variables should, obviously, be eliminated, and missing variables should be added. Deciding which set of attributes best describe the object and/or which attributes are essential to make the representation of the object as close as possible to the real world situation is a task that requires a thorough working knowledge of the problem space.

Look for variables that are common to groups of objects (classes). These commonalities are fundamental to the creation of the class hierarchy.

Look for variables that have the same value for all objects which are instances of the same class. Variables that have the same value for all objects of a class are potential class variables. Variables that do not change in value for all objects of a class may be treated as constants. Analyze constants to determine whether they should be implemented as variables or as methods which simply return the constant value.

Look for variables that can be calculated or derived from other variables. These variables may be replaced by methods which calculate them.

Look for variables that can be decomposed into more elementary variables. These variables may themselves be defined as individual objects. Objects which contain such variables are potential composite objects.

Look for variables that are defined for only one class. Classes that contain these variables may be classes without subclasses in the class hierarchy.

2.1.3 ANALYSIS OF THE OBJECT'S METHODS

The purpose of this step is to analyze the methods defined for the objects which resulted from step 2.1.1 and list observations that may be important in the construction of the class hierarchy. The following guidelines are used during this phase:

Look for methods that are common to several classes. Commonalities of methods are fundamental to the creation of the class hierarchy.

Every concrete class should have, as a minimum, a set of methods whose purpose is to create, delete, maintain, and display the instances (objects) of that class. Note, however, that abstract classes do not need these methods as they do not have any instances.

In order to enforce encapsulation, it may also be necessary to define methods for accessing and updating each variable.

2.2 REFINEMENT OF THE OBJECTS AND CLASSES

2.2.1 ADDITION OF NECESSARY INFORMATION

With the basis in the problem space, analyze the objects to make sure that all necessary information can be obtained through the set of methods defined so far. If not, then add the necessary variables/methods.

2.2.2 ELIMINATION OF REDUNDANT INFORMATION

Eliminate variables that can be derived from other variables. This can be done by defining a method that calculates the desired value using the information provided by other variables. It is not necessary to define a variable to store information that can be calculated.[3]

Note that in the case of 'A derives B' and 'B derives A', one may have to look to the problem space to determine whether it is A or B that should be stored as an actual variable. Also remember that having the two variables A and B defined for an object (class) at this point in the design simply means that the object is responsible for providing this information upon request - it is even possible that neither variable will actually be a part of the final design.

[3]It should be realized that this is a general "rule of thumb". For efficiency considerations, it may be necessary to actually have both variables in the object.

2.2.3 DETERMINATION OF CLASS AND INSTANCE VARIABLES

A variable that has the same value for all objects (instances) of a class can be defined a class variable. Those variables whose values vary from object to object are defined to be instance variables.

Some class variables may not be expected to change in value, in which case they may be more properly considered to be constants. These constant values could be more simply provided by methods which return the constant value without checking the value of any variable. Alternatively, these kind of variables could also be defined as class variables. In most cases, it would probably be more appropriate to define constants via methods rather than variables.

2.2.4 IDENTIFICATION OF COMPOSITE OBJECTS

Define as individual objects those variables that can be decomposed into more elementary variables. This is an iterative step in that the newly defined objects may themselves have variables that can be further decomposed into more elementary variables. This process continues until the object's variables can no longer be decomposed.

2.3 ORGANIZATION OF THE CLASSES INTO A HIERARCHY

2.3.1 ANALYSIS OF THE IMPLEMENTATION LANGUAGE

The design of the class hierarchy depends directly on the facilities provided by the implementation language or OODBMS used. In this phase, the following questions should be answered:

Does the system provide single, multiple, or selective inheritance?

If the system provides multiple inheritance, what are its conflict resolution rules (i.e., the way in which the system handles the conflicts that may arise when the names of variables and methods are used more than once)?

Can methods inherited from a superclass be overridden (redefined) in the subclasses?

Can variables inherited from a superclass be overridden (redefined) in the subclasses?

Does the system provide single, multiple, or selective inheritance? The form of inheritance provided by the system has a direct influence on the design of the class hierarchy. A design utilizing multiple inheritance will most likely be quite different from a design utilizing only single inheritance.

If the system provides multiple inheritance, what are its conflict resolution rules? Multiple inheritance, even though giving more flexibility to the design, should be used carefully as conflicts involving the names of variables and methods may occur. That is, problems may arise if two or more superclasses have variables/methods with the same name (see [Nels90, NMO91] for more information on this problem).

Can methods inherited from a superclass be overridden (redefined) in the subclasses? Most systems do allow the redefinition of inherited methods in subclasses. Redefinition permits a method to be modified or customized to meet the specific needs of the subclass, while keeping the same name as that found in the superclass.

Can variables inherited from a superclass be overridden (redefined) in the subclasses? Some OOP languages do not allow the redefinition of inherited variables in subclasses. That is, a subclass cannot have a variable with the same name as an inherited variable. This results in a name conflict and, consequently, in a compilation error. Other OOP languages, however, do permit variables to be renamed in subclasses. Name conflicts are avoided in this case by using the locally defined version.

2.3.2 CONSTRUCTION OF THE HIERARCHIES

The following guidelines for the analysis of inheritance relationships between classes have been proposed [WWW90], and they have (for the most part) been adopted in our design methodology:

Model a "kind-of" hierarchy.

Factor common methods as high as possible.

Do not allow abstract classes to inherit from concrete classes.

Eliminate classes that do not add functionality.

Model a "kind-of" hierarchy. A "kind-of" hierarchy means that every class should be a specific kind of its superclasses. Subclasses should support all of the methods and variables defined by their superclasses and possibly more. Ensuring that this is so will make the classes more reusable as it becomes easier to see where, in an existing hierarchy, a new class should be placed.

When a subclass would include only some of the methods and/or variables defined by its superclasses, it is good practice to create an abstract class with all of the methods and variables common to both classes. The hierarchy is then restructured in such a way that both the class and its former superclass become subclasses of the newly created abstract (super)class.

Factor common methods as high as possible. Factoring common methods as high as possible in the hierarchy is based on the principle that if a set of classes all support a common method and/or variable, then they should all inherit that method/variable from a common superclass. If a common superclass does not already exist then create one, and move all of the common methods/variables to it. This new superclass will more than likely be an abstract class.

Do not allow abstract classes to inherit from concrete classes. Abstract classes, by their very nature, support their methods and/or variables in implementation-independent ways. Concrete classes, on the other hand, may specifically depend upon implementation. In the case where an abstract class does inherit from a concrete class, it is still possible to achieve this goal by creating another abstract class from which both the abstract and concrete classes can inherit their common methods/variables.

Eliminate classes that do not add functionality. Classes which have no methods should ordinarily be discarded. However, if a class inherits a method that it will implement in some unique way, then it adds functionality in spite of having no methods of its own, and should therefore be kept.

It has also been said that abstract classes which define no methods have no use, and they should therefore be eliminated [WWW90]. This is where we differ in this step from [WWW90]. Any class, even though defining no methods of its own, can still have variables that are used to store important information. Therefore, only those classes which add neither functionality nor store any information should be discarded.[4]

2.3.3 REVIEW OF THE CLASSES' VARIABLES/METHODS

During the construction of the class hierarchies, achieving a perfect match among the classes' variables and methods is highly unlikely. For example, when a class is defined as a subclass of another class, the subclass inherits all of the variables and methods of its superclass. However, the subclass may not actually need all of the inherited variables/methods. If the system allows selective inheritance, then specific variables/methods can be excluded, but if the system does not allow selective inheritance then some classes will have to be modified.

These changes in the definition of certain classes are necessary for the achievement of similarity among classes. Variables/methods that are not strictly necessary could be added to a class in order to 'force' it to fit into the set of subclasses of a given class. This seems to contradict the philosophy of step described in section 2.2.2 (Elimination of Redundant Information). However, the purpose of these changes in the classes' variables/methods is to allow the design of the simplest class hierarchy possible while still keeping the semantics inherent to the problem space.

However, the hierarchy of classes should also reflect the real world situation. That is, a contrived class hierarchy should not be created just for the sake of capturing certain commonalities [CY90]. Therefore, the variables/methods added to a class must also be meaningful in terms of the problem space.

3 THE DESIGN OF THE TACTICAL DATABASE

A Combat Direction System (CDS) consists of a complex set of data inputs, user consoles, converters, adapters, and radio terminals interconnected with high speed computers and their stored programs. Combat data is collected, processed, and composed into a picture of the overall tactical situation that enables the force commander to make rapid and accurate decisions. [DON85]

The Low Cost Combat Direction System (LCCDS) [dePa90, Ross89, SS90], sponsored by the Naval Sea Systems Command (NAVSEA), will implement the basic features of a CDS on commercially available microprocessor-based workstations. The system is intended for use on board non-combatant ships, where no automated combat information processing capability currently exists. The Tactical Database (TDB) is that part of the LCCDS which contains information necessary to establish an accurate picture of a ship's environment.

[4]It should be noted, however, that [WWW90] represents a strictly responsibility-driven approach; therefore, no variables have been defined as of yet. Since we incorporate variables at a much earlier stage of our design, it is possible to have meaningful classes which have variables only (i.e., they have no methods).

3.1 IDENTIFICATION OF THE OBJECTS AND CLASSES

3.1.1 INITIAL DEFINITION OF THE OBJECTS AND CLASSES

Normally, this phase would start with an in-depth analysis of the problem space so that the designer could make a list of potential objects and/or classes. In this case, the initial phase of the design consisted basically of the analysis of the set of objects proposed in [SS90], as most of the objects were readily identified therein.

The minimum set of objects (classes) that the TDB should contain has been specified as follows [SS90]:

CLASS: OWNSHIP

Variables:	GEOGRAPHICAL POSITION TIME OF POSITION COURSE SPEED ORIGIN TRACK NUMBER TYPE TRACK HISTORY
Methods:	MONITOR OWNSHIP POSITION NAVIGATIONAL COMPUTATION CPA PROCESSING

Description: An object of class OWNSHIP represents the ship that is operating the LCCDS. It therefore follows that there should be only a single instance of OWNSHIP for each Tactical Database.

CLASS: TENTATIVE TRACK

Variables:	GEOGRAPHICAL POSITION RELATIVE POSITION TIME OF POSITION COURSE SPEED CATEGORY IDENTITY ORIGIN TRACK NUMBER TYPE
Methods:	NEW TRACK ESTABLISHMENT TRACK POSITION DATA TRACK COURSE AND SPEED DETERMINATION DEAD RECKONING INITIAL CATEGORY ASSIGNMENT INITIAL IDENTITY ASSIGNMENT TRACK TERMINATION

Description: An object of class TENTATIVE TRACK represents a new track generated by local sensors (radar). Once a valid course and speed is established for it, the Tentative Track will become a Firm Track. This can happen as a result of any of the following conditions: (1) After three manual position updates by the user; (2) Category entered manually before three position updates; or (3) Manually declared firm by the user.

CLASS: REFERENCE POINT, NAVIGATION HAZARD, and MAN IN WATER

Variables: GEOGRAPHICAL POSITION
RELATIVE POSITION
TIME OF POSITION
CATEGORY
IDENTITY
ORIGIN

Methods: CPA PROCESSING

Description: An object of class REFERENCE POINT represents a fixed point on the surface of the Earth which is used as a reference. An object of class NAVIGATION HAZARD represents a point on the surface which might be hazardous to navigation (icebergs, shallow waters, reefs, mines, etc) and therefore must be avoided. An object of class MAN IN WATER represents a point on the surface of the water on which a man (or a group of men) is supposed to be.

CLASS: DATA LINK REFERENCE POINT

Variables: GEOGRAPHICAL POSITION
TIME OF POSITION
CATEGORY
IDENTITY
ORIGIN

Methods: CPA PROCESSING

Description: An object of class DATA LINK REFERENCE POINT represents a fixed geographic reference position common to all Link 11 participating units (a Link 11 unit is a source of remote information).

CLASS: FORMATION CENTER and POSITION AND INTENDED MOVEMENT

Variables: GEOGRAPHICAL POSITION
TIME OF POSITION
RELATIVE POSITION
COURSE
SPEED
CATEGORY
IDENTITY
ORIGIN

Methods: DEAD RECKONING
CPA PROCESSING

Description: An object of class FORMATION CENTER represents a moving geographic position representing the center of a group of ships steaming in formation. An object of class POSITION AND INTENDED MOVEMENT represents the planned position of Ownship or the formation based on a pre-computed base course and speed to arrive at a destination at the required time.

CLASS: AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK

Variables: GEOGRAPHICAL POSITION
RELATIVE POSITION
TIME OF POSITION
COURSE
SPEED
HEIGHT (defined only for AIR TRACK)
DEPTH (defined only for SUBSURFACE TRACK)
CATEGORY
IDENTITY
ORIGIN
TRACK NUMBER
TYPE

Methods: CPA PROCESSING
TRACK COURSE AND SPEED DETERMINATION
DEAD RECKONING
TRACK POSITION PREDICTION
TRACK HISTORY PROCESSING
MANUAL TERMINATION
IDENTIFICATION FUNCTION

Description: An object of class AIR TRACK represents a real-world object which is in the air and, consequently, has HEIGHT as one of its attributes to indicate its current altitude. An object of class SURFACE TRACK represents a real-world object which is on the surface. An object of the class SUBSURFACE TRACK represents a real-world object which is underwater and, consequently, has DEPTH as one of its attributes.

CLASS: WAYPOINT

Variables: GEOGRAPHICAL POSITION
RELATIVE POSITION
TIME OF POSITION
STEAMING ROUTE
CATEGORY
IDENTITY
ORIGIN

Methods: WAYPOINT GEOMETRY

Description: An object of class WAYPOINT represents a destination point on the surface. Each waypoint can be viewed as a "node" in a determined route.

3.1.2 ANALYSIS OF THE OBJECTS' VARIABLES

Following is a summary of observations about the objects' variables that were considered important to the construction of the class hierarchy:

GEOGRAPHICAL POSITION, TIME OF POSITION, and ORIGIN are common to all classes.

CATEGORY and IDENTITY are common to all classes except OWNSHIP.

The classes TENTATIVE TRACK, SURFACE TRACK, AIR TRACK, and SUBSURFACE TRACK all have similar variables.

The classes REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, DATA LINK REFERENCE POINT, FORMATION CENTER, and POSITION AND INTENDED MOVEMENT all have similar variables.

GEOGRAPHICAL POSITION, TIME OF POSITION, and RELATIVE POSITION can each be decomposed into more elementary variables.

GEOGRAPHICAL POSITION and RELATIVE POSITION are not independent (i.e., either one can be calculated from the other).

3.1.3 ANALYSIS OF THE OBJECTS' METHODS

Following are observations about the objects' methods that were considered important to the construction of the class hierarchy:

CPA PROCESSING is common to all classes except TENTATIVE TRACK and WAYPOINT.

The classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK all have the same methods.

The classes REFERENCE POINT, NAVIGATION HAZARD, MAN IN WATER, DATA LINK REFERENCE POINT, POSITION AND INTENDED MOVEMENT, and FORMATION CENTER all have the same methods, with the exception of DEAD RECKONING.

3.2 REFINEMENT OF THE OBJECTS AND CLASSES

3.2.1 ADDITION OF NECESSARY INFORMATION

The methods CREATE INSTANCE, DELETE INSTANCE, UPDATE INSTANCE, and DISPLAY INSTANCE were added to all of the classes to ensure that each concrete class has a set of methods for creating, deleting, maintaining, and displaying its instances (objects).

3.2.2 ELIMINATION OF REDUNDANT INFORMATION

The main result of this step was the decision not to include both the variables GEOGRAPHICAL POSITION and RELATIVE POSITION together in the definition of a class. When either variable was defined for a certain class, the other was then calculated by a method.

3.2.3 DETERMINATION OF CLASS AND INSTANCE VARIABLES

It turns out that every class "variable" in our system was actually a class "constant" (i.e., not only did it have the same value for all instances of a class, that value is not expected to ever change at run time). The decision was made to define a method to return a constant value rather than defining a class variable in all cases.

3.2.4 IDENTIFICATION OF COMPOSITE OBJECTS

The analysis of the objects' variables, showed that each of the variables GEOGRAPHICAL POSITION, RELATIVE POSITION, and TIME OF POSITION can be decomposed into more elementary variables.

GEOGRAPHICAL POSITION has two components: latitude and longitude. Latitude and longitude, in turn, each have two components: angle and hemisphere (north/south or east/west). The component angle can be further divided into three components: degree, minute, and second.

Therefore, it is possible to create three new classes: ANGLE; LATITUDE; and LONGITUDE. These objects are defined as follows:

CLASS: ANGLE

Superclasses:	None
Class Variables:	None
Instance Variables:	DEGREE: (0..359) [*0*] MINUTE: (0..59) [*0*] SECOND: (0..59) [*0*]
Methods:	GET & SET DEGREE GET & SET MINUTE GET & SET SECOND

CLASS: LATITUDE

Superclasses:	None
Class Variables:	None
Instance Variables:	LATITUDE ANGLE: ANGLE [*0:0:0*] LATITUDE HEMISPHERE: {N, S} [*N*]
Methods:	GET & SET LATITUDE ANGLE GET & SET LATITUDE HEMISPHERE

CLASS: LONGITUDE

Superclasses:	None
Class Variables:	None
Instance Variables:	LONGITUDE ANGLE: ANGLE [*0:0:0*] LONGITUDE HEMISPHERE: {E, W} [*W*]
Methods:	GET & SET LONGITUDE ANGLE GET & SET LONGITUDE HEMISPHERE

The variable relative position has two components: bearing and range. A bearing (or direction heading) can be further divided into an angle and a reference north (true or magnetic). Two more classes can therefore be created: DIRECTION and RELATIVE POSITION. These classes are defined as follows:

CLASS: DIRECTION

Superclasses:	None
Class Variables:	None
Instance Variables:	DIRECTION ANGLE: ANGLE [*0:0:0*] REFERENCE NORTH: {T, M} [*T*]
Methods:	GET & SET DIRECTION ANGLE GET & SET REFERENCE NORTH

CLASS: RELATIVE POSITION

Superclasses:	None
Class Variables:	None
Instance Variables:	REL POS BEARING: DIRECTION [*0:0:0:T*] RANGE: real [*0.0*]
Methods:	GET & SET REL POS BEARING GET & SET REL POS RANGE

The variable time of position has four simple components: hour, minute, second, and time zone. Another class, TIME, was then defined:

CLASS: TIME

Superclasses:	None
Class Variables:	None
Instance Variables:	HOUR: (00..23) [*00*] MINUTE: (00..59) [*00*] SECOND: (00..59) [*00*] TIME ZONE: (A..Z) [*Z*]
Methods:	GET & SET HOUR GET & SET MINUTE GET & SET SECOND GET & SET TIME ZONE

3.3 ORGANIZATION OF THE CLASSES INTO A HIERARCHY

3.3.1 ANALYSIS OF THE IMPLEMENTATION LANGUAGE

Based on the assumption that the Gemstone Database Management System [KBCG89, Serv89] is the system that best fits the requirements of the LCCDS Project [Ross89], the following facts should be taken into consideration during the construction of the class hierarchy:

Only single inheritance is allowed.

Methods inherited from a superclass can be redefined in subclasses.

Variables inherited from a superclass can be redefined in subclasses, as long as the constraint on the variables defined in the subclass are the same as or a subset of the constraint in the superclass.

3.3.2 CONSTRUCTION OF THE HIERARCHIES

As previously mentioned, the similarities of methods/variables among classes is the fundamental factor in the construction of the class hierarchies. However, it is important that the hierarchies should represent the real world (problem space) as closely as possible. The resulting hierarchy (or hierarchies) should not just be a number of similar classes organized in an otherwise meaningless hierarchy.

In the design process, the similarities of methods/variables among classes were listed in steps 3.1.2 (2.1.2) and 3.1.3 (2.1.3). Also, new methods and variables were defined during step 3.2 (2.2). The resulting set of classes can then be organized into a hierarchy (or hierarchies).

It was observed that each of the classes defined for the Tactical Database represents either a point (fixed or nonfixed position) or an area (region) in the LCCDS scenario. This suggests a division of the classes into two groups: points and areas. These two groups of classes were named, respectively, TDB POINT (Tactical Database Point) and TDB AREA (Tactical Database Area). Considering that all classes represent real world objects, these two classes were defined as subclasses of the class TDB OBJECT (Tactical Database Object). This initial class hierarchy is shown in Figure 2.

The analysis of the commonalities among the classes AIR TRACK, SURFACE TRACK, SUBSURFACE TRACK, and TENTATIVE TRACK showed that all of these classes could be defined as subclasses of a class TRACK.

While the classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK represent, respectively, objects that are in the air, on the surface, and

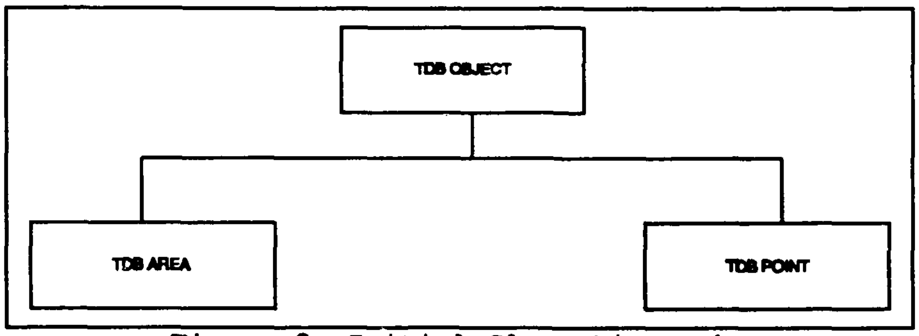

Figure 2: Initial Class Hierarchy

underwater, the class TENTATIVE TRACK represents an object which can be in any of these categories. Objects of classes AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK are considered firm tracks because their positions can be updated by the system at regular intervals of time. On the other hand, an object of class TENTATIVE TRACK only becomes a firm track after a valid course and speed are established for it.

In terms of the real world situation, it is meaningful then to create the class TRACK with two subclasses: FIRM TRACK and TENTATIVE TRACK. The class FIRM TRACK, in turn, is divided into three subclasses: AIR TRACK, SURFACE TRACK, and SUBSURFACE TRACK.

Note also that the class OWNSHIP, even though not representing an object that is being tracked by the ship's sensors, can be considered as a particular case of the class SURFACE TRACK for which the relative position is equal to zero (the relative position of all other objects is calculated with respect to Ownship). The class OWNSHIP was, therefore, defined as a subclass of the class SURFACE TRACK.

Thus, considering the class TRACK as the root, the hierarchy of classes shown in Figure 3 was defined.

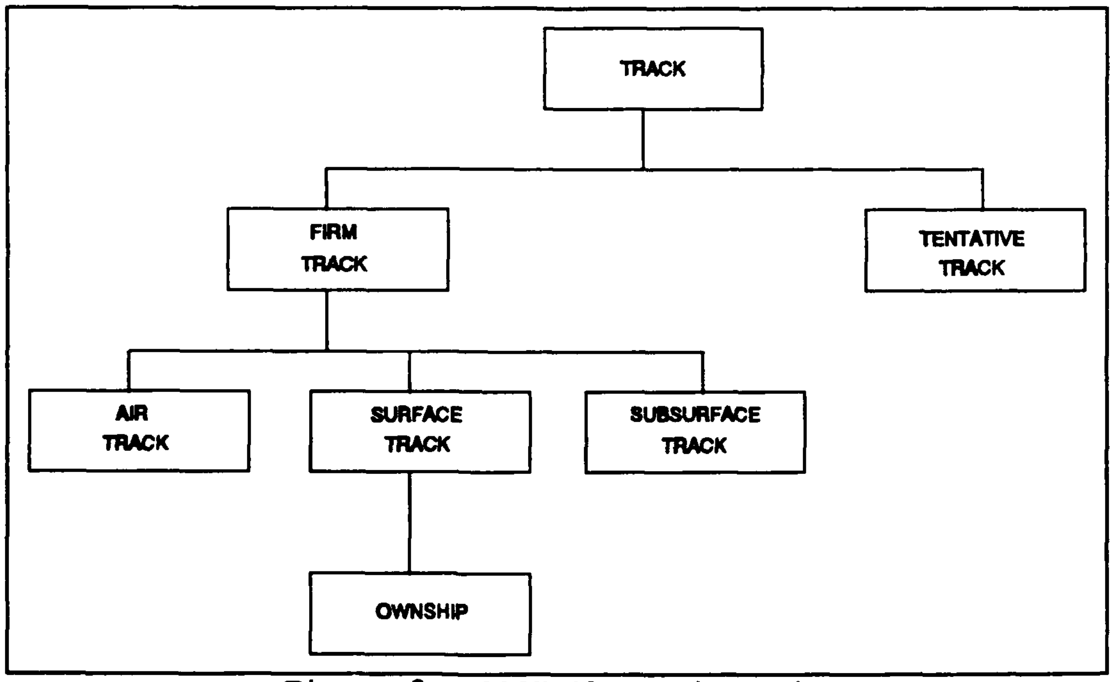

Figure 3: TRACK Class Hierarchy

The analysis of the commonalities among the classes REFERENCE POINT, DATA LINK REFERENCE POINT, WAYPOINT, FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT showed that these classes can be defined as subclasses of a class SPECIAL POINT. However, the variables COURSE and SPEED were not defined for the classes REFERENCE POINT, WAYPOINT, and DATA LINK REFERENCE POINT (i.e, these points are fixed). Based on this difference, these classes can be divided into two groups: fixed and nonfixed.

Therefore, two subclasses of the class SPECIAL POINT were created: FIXED SPECIAL POINT and NONFIXED SPECIAL POINT. The classes REFERENCE POINT, DATA LINK REFERENCE POINT, and WAYPOINT were defined as subclasses of FIXED SPECIAL POINT and the classes FORMATION CENTER, NAVIGATION HAZARD, MAN IN WATER, and POSITION AND INTENDED MOVEMENT were defined as subclasses of NONFIXED SPECIAL POINT.

Therefore, considering SPECIAL POINT as the root, the hierarchy of classes shown in Figure 4 was defined.

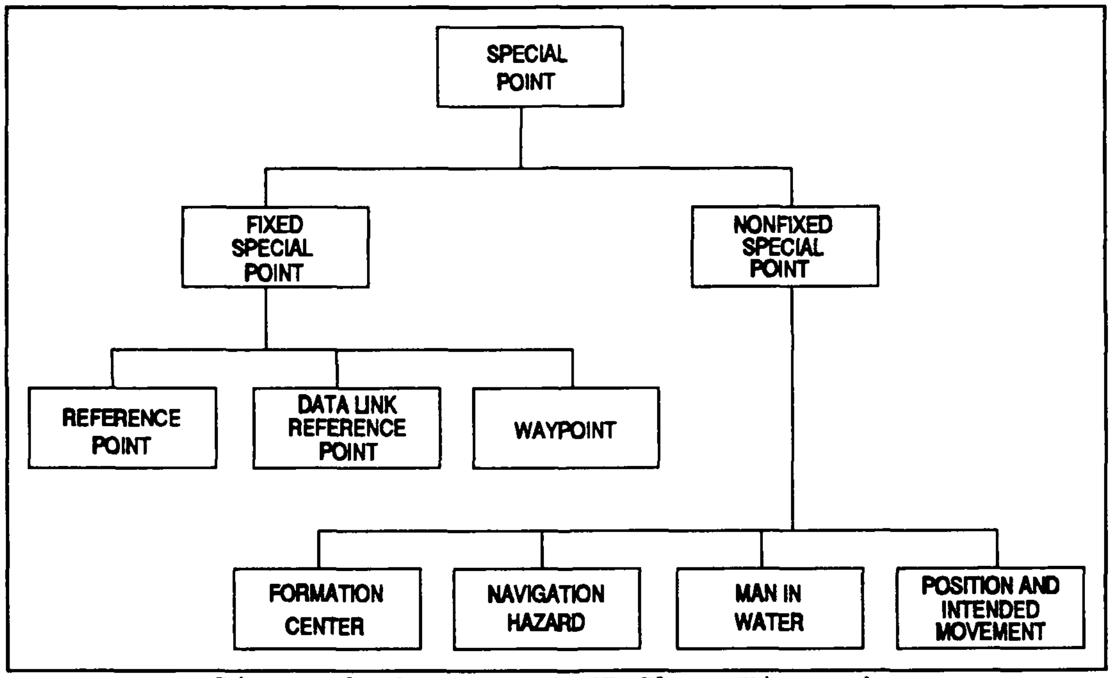

Figure 4: SPECIAL POINT Class Hierarchy

Since the classes SPECIAL POINT and TRACK both represent points in the LCCDS scenario, they were each defined as subclasses of the class TDB POINT. Also, since TDB POINT was defined as a subclass of the class TDB OBJECT, the entire hierarchy is now complete. This final hierarchy is shown in Figure 5.

3.3.3 REVIEW OF THE CLASSES' VARIABLES/METHODS

Following are some of the more important changes which were made in the definition of various classes in order to make them fit into the class hierarchy created (see [dePa90] for all of the changes):

The variable TDB OBJECT NUMBER was added to the definition of all classes, in order to identify uniquely every instance of each class. The variable TRACK NUMBER was, consequently, eliminated in the classes

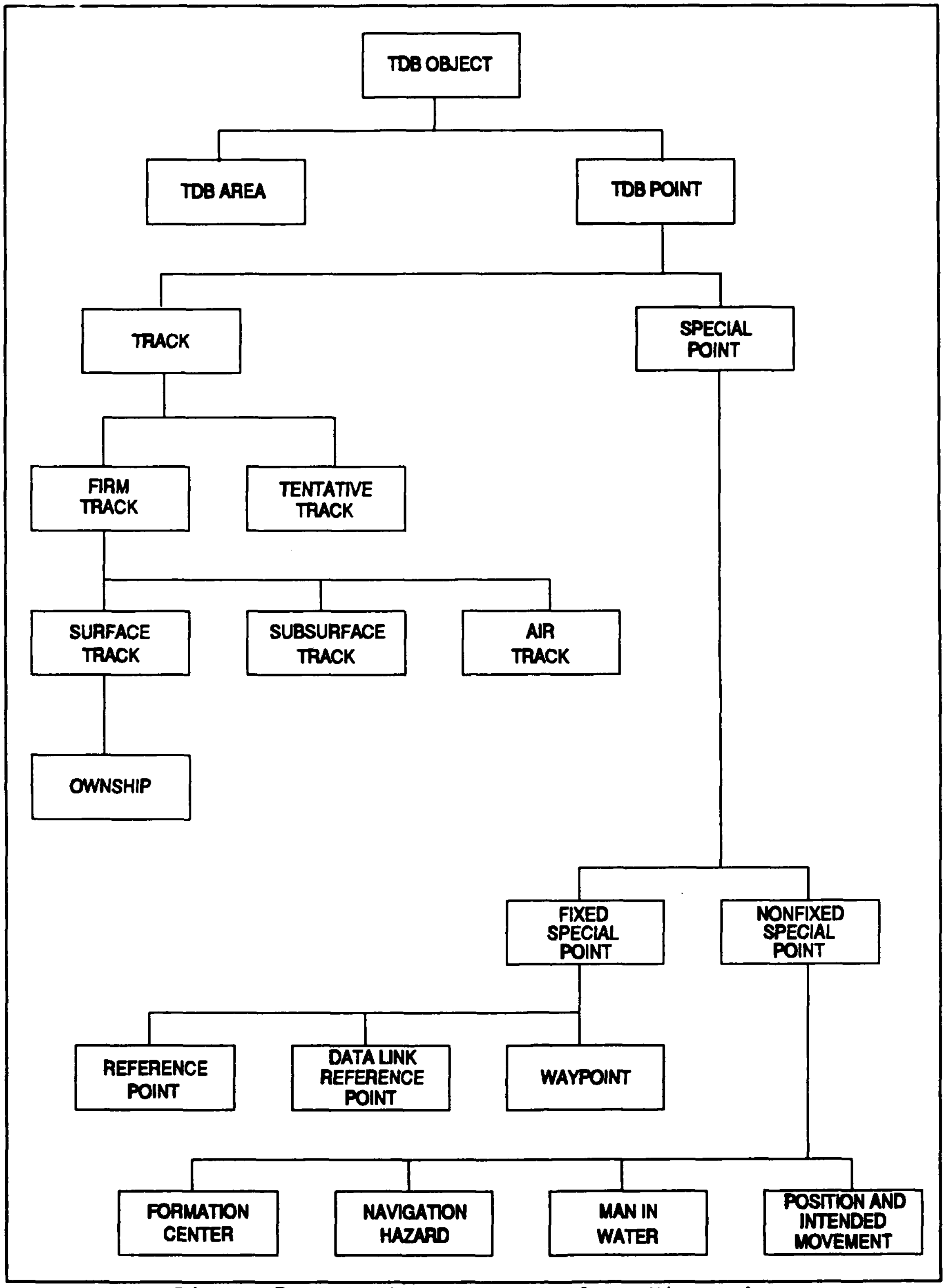

Figure 5: TACTICAL DATABASE Class Hierarchy

for which it was originally defined as it is now unnecessary.[5] The method CPA PROCESSING was added to the definition of the classes TENTATIVE TRACK and WAYPOINT. With this addition, CPA PROCESSING became common to all classes.

The variables RELATIVE POSITION, ORIGIN, and IDENTITY were added to the class OWNSHIP. These variables were assigned the constant values *0:0:0:T:0.0*, *Local Manual*, and *Friendly*, respectively.

The variable ORIGIN was redefined for the class SPECIAL POINT, being allowed to assume only the values *Local Manual* and *Remote.*

4 FINAL DEFINITION OF THE CLASSES

All of the classes for the Tactical Database may now be defined as follows:

CLASS: ANGLE

Superclasses:	None
Class Variables:	None
Instance Variables:	DEGREE: (0..359) [*0*] MINUTE: (0..59) [*0*] SECOND: (0..59) [*0*]
Methods:	GET & SET DEGREE GET & SET MINUTE GET & SET SECOND

CLASS: LATITUDE

Superclasses:	None
Class Variables:	None
Instance Variables:	LATITUDE ANGLE: ANGLE [*0:0:0*] LATITUDE HEMISPHERE: {N, S} [*N*]
Methods:	GET & SET LATITUDE ANGLE GET & SET LATITUDE HEMISPHERE

CLASS: LONGITUDE

Superclasses:	None
Class Variables:	None
Instance Variables:	LONGITUDE ANGLE: ANGLE [*0:0:0*] LONGITUDE HEMISPHERE: {E, W} [*W*]
Methods:	GET & SET LONGITUDE ANGLE GET & SET LONGITUDE HEMISPHERE

[5] It should be noted that anytime a variable or method name is changed in order to simplify the class hierarchy, it could cause problems for users of the system if they are used to an existing system (manual or automated) which utilizes the original name. This particular problem is, however, very easy to rectify. Simply provide additional methods (using the original name) which in turn call the methods with the new names or access the renamed variable (e.g., the value of the variable TDB OBJECT NUMBER could also be available via the methods GET & SET TRACK NUMBER).

CLASS: DIRECTION

Superclasses:	None
Class Variables:	None
Instance Variables:	DIRECTION ANGLE: ANGLE [*0:0:0*] REFERENCE NORTH: {T, M} [*T*]
Methods:	GET & SET DIRECTION ANGLE GET & SET REFERENCE NORTH

CLASS: RELATIVE POSITION

Superclasses:	None
Class Variables:	None
Instance Variables:	REL POS BEARING: DIRECTION [*0:0:0:T*] RANGE: real [*0.0*]
Methods:	GET & SET REL POS BEARING GET & SET REL POS RANGE

CLASS: TIME

Superclasses:	None
Class Variables:	None
Instance Variables:	HOUR: (00..23) [*00*] MINUTE: (00..59) [*00*] SECOND: (00..59) [*00*] TIME ZONE: (A..Z) [*Z*]
Methods:	GET & SET HOUR GET & SET MINUTE GET & SET SECOND GET & SET TIME ZONE

CLASS: TDB OBJECT

Superclasses:	None
Class Variables:	None
Instance Variables:	TDB OBJECT NUMBER: integer [*0*] TIME OF POSITION: TIME [*00:00:00:Z*] ORIGIN: {Local Manual, Local Auto, Remote} [*Local Manual*]
Methods:	MENU CREATE INSTANCE DELETE INSTANCE UPDATE INSTANCE DISPLAY INSTANCE CPA PROCESSING GET & SET TDB OBJECT NUMBER GET & SET TIME OF POSITION GET & SET ORIGIN

CLASS: TDB AREA

Superclasses:	TDB OBJECT
Class Variables:	None
Instance Variables:	CENTER GEO POSITION: GEOGRAPHICAL POSITION *[0:0:0:N]* NORTH LIMIT: GEOGRAPHICAL POSITION *[0:0:0:N]* SOUTH LIMIT: GEOGRAPHICAL POSITION *[0:0:0:S]* EAST LIMIT: GEOGRAPHICAL POSITION *[0:0:0:E]* WEST LIMIT: GEOGRAPHICAL POSITION *[0:0:0:W]* IDENTITY: {Hot, Protected, Unprotected} *[Unprotected]* COURSE: ANGLE *[0:0:0]* SPEED: real *[0.0]*
Methods:	DEAD RECKONING DRAW AREA GET & SET CENTER GEO POSITION GET & SET NORTH LIMIT GET & SET SOUTH LIMIT GET & SET EAST LIMIT GET & SET WEST LIMIT GET & SET IDENTITY GET & SET COURSE GET & SET SPEED

CLASS: TDB POINT

Superclasses:	TDB OBJECT
Class Variables:	None
Instance Variables:	None
Methods:	PLOT TDB POINT

CLASS: TRACK

Superclasses:	TDB POINT
Class Variables:	None
Instance Variables:	RELATIVE POSITION: RELATIVE POSITION *[0:0:0:T:0.0]* COURSE: ANGLE *[0:0:0]* SPEED: real *[0.0]* IDENTITY: {Unknown, Friendly, Hostile} *[Unknown]*
Methods:	GET & SET RELATIVE POSITION GET & SET COURSE GET & SET SPEED GET & SET IDENTITY CALCULATE RELATIVE POSITION TRACK COURSE AND SPEED DETERMINATION DEAD RECKONING

CLASS: TENTATIVE TRACK

Superclasses:	TRACK
Class Variables:	None
Instance Variables:	CATEGORY: {Air, Surface, Subsurface} [*Air*]
Methods:	NEW TRACK ESTABLISHMENT INITIAL CATEGORY ASSIGNMENT INITIAL IDENTITY ASSIGNMENT GET & SET CATEGORY

CLASS: FIRM TRACK

Superclasses:	TRACK
Class Variables:	None
Instance Variables:	None
Methods:	IDENTIFICATION FUNCTION TRACK HISTORY PROCESSING

CLASS: AIR TRACK

Superclasses:	FIRM TRACK
Class Variables:	None
Instance Variables:	HEIGHT: real [*0.0*]
Methods:	GET CATEGORY GET & SET HEIGHT

CLASS: SURFACE TRACK

Superclasses:	FIRM TRACK
Class Variables:	None
Instance Variables:	None
Methods:	GET CATEGORY

CLASS: SUBSURFACE TRACK

Superclasses:	FIRM TRACK
Class Variables:	None
Instance Variables:	DEPTH: real [*0.0*]
Methods:	GET CATEGORY GET & SET DEPTH

CLASS: OWNSHIP

Superclasses:	SURFACE TRACK
Class Variables:	None
Instance Variables:	GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [*0:0:0:N*] GREENWICH MEAN TIME: TIME [*00:00:00:Z*] JULIAN DATE: (0001..9366) [*0001*]
Methods:	MONITOR OWNSHIP POSITION GET & SET GREENWICH MEAN TIME GET & SET JULIAN DATE

CLASS: SPECIAL POINT

Superclasses:	TDB POINT
Class Variables:	None
Instance Variables:	GEOGRAPHICAL POSITION: GEOGRAPHICAL POSITION [*0:0:0:N*] ORIGIN: {Local Manual, Remote} [*Local Manual*]
Methods:	GET & SET GEOGRAPHICAL POSITION GET & SET ORIGIN CALCULATE RELATIVE POSITION GET CATEGORY

CLASS: FIXED SPECIAL POINT

Superclasses:	SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	None

CLASS: REFERENCE POINT

Superclasses:	FIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY

CLASS: DATA LINK REFERENCE POINT

Superclasses:	FIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY

CLASS: WAYPOINT

Superclasses:	FIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	STEAMING ROUTE: (1..6) [*1*]
Methods:	WAYPOINT GEOMETRY GET & SET STEAMING ROUTE GET ORIGIN GET IDENTITY

CLASS: NONFIXED SPECIAL POINT

Superclasses:	SPECIAL POINT
Class Variables:	None
Instance Variables:	COURSE: ANGLE [*0:0:0*] SPEED: real [*0.0*]
Methods:	DEAD RECKONING GET & SET COURSE GET & SET SPEED

CLASS: FORMATION CENTER

Superclasses:	NONFIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY

CLASS: NAVIGATION HAZARD

Superclasses:	NONFIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY

CLASS: MAN IN WATER

Superclasses:	NONFIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY

CLASS: POSITION AND INTENDED MOVEMENT

Superclasses:	NONFIXED SPECIAL POINT
Class Variables:	None
Instance Variables:	None
Methods:	GET IDENTITY GET ORIGIN

5 CONCLUSIONS

The primary goal of our design methodology is to provide a simple and systematic way of approaching the problems of object-oriented design. Even though the steps that form the methodology were proposed in a certain logical order, they are flexible in the sense that the order in which they are performed can be changed according to the designer's preference. The iterative nature of the design process is evidence that the order in which each step is accomplished is fairly flexible.

Presently, various object-oriented languages provide quite different facilities and features. This explains the necessity of an analysis of the implementation language prior to the construction of the class hierarchies. However, some of these facilities and features are fairly typical of object-oriented languages in general (i.e., they are provided by nearly all of them). Thus, in order to produce a design that is independent of the specific system used, the following should be considered:

All object-oriented languages allow single inheritance[6], but not all allow multiple inheritance or selective inheritance.

[6]An "object-oriented" language which does not provide inheritance is more appropriately considered to be "object-based" [Wegn87].

The redefinition of inherited methods by the subclasses of a class is allowed in nearly all object-oriented languages.

The redefinition of inherited variables by the subclasses of a certain class is not allowed in all languages.

Not all object-oriented languages (including those provided by OODBMSs) allow run-time changes in the definition of classes or in the class hierarchy.

It is expected, however, that these differences will eventually disappear with the evolution of object-oriented languages, giving the designer more freedom to pursue a language-independent design.

REFERENCES

[Booc91] G. Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Co, Menlo Park, CA, 1991.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press (Prentice Hall), Englewood Cliffs, NJ, 1990.

[dePa90] E.G. de Paula. *A Tactical Database for the Low Cost Combat Direction System.* Master's Thesis, Naval Postgraduate School, Monterey, CA, Dec 1990.

[DON85] Department of the Navy. *Systems Engineering Handbook, Volume I: Combat Direction System Model 5.* NAVSEA Technical Report No 0967-LP-027-8602, Feb 1985.

[KBCG89] W. Kim, N. Ballou, H-T. Chou, J.F. Garza, and D. Woelk. "The GemStone Data Management System", in *Object-Oriented Concepts, Databases, and Applications,* edited by W. Kim and F.H. Lochovsky, ACM Press (Addison-Wesley Publishing Co), New York, NY, 1989, pp 283-308.

[Nels90] M.L. Nelson. *An Introduction to Object-Oriented Programming.* Naval Postgraduate School, Monterey, CA, Technical Report No NPS52-90-024, Apr 1990.

[NMO91] M.L. Nelson, J.M. Moshell, A. Orooji. "The Case For Encapsulated Inheritance", *Proceedings of the 24th Annual Hawaii International Conference on System Sciences (HICSS-24), Vol II: Software Technology,* Jan 1991, Koloa, HI, pp 219-227.

[Ross87] R. Ross. *Entity modeling: Techniques and Applications.* Database Research Group, Boston, MA, 1987.

[Ross89] D.L. Ross. *Object-Oriented Database Manager for the Low Cost Combat Direction System.* Master's Thesis, Naval Postgraduate School, Monterey, CA, Dec 1989.

[SB86] M. Stefik and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations", *The AI Magazine,* Vol 6, No 4, Winter 1986, pp 40-62.

[Serv89] Servio Logic Corp. *Programming in OPAL.* Servio Logic Corp, Beaverton, OR, 1989.

[SM88] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[SS90] J. Seveney and G. Steinberg. *Requirements Analysis for a Low Cost Combat Direction System.* Master's Thesis, Naval Postgraduate School, Monterey, CA, Jun 1990.

[Ullm82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1982.

[Wegn87] P. Wegner. "Dimensions of Object-Based Language Design", *OOPSLA'87 Proceedings,* Oct 1987, Orlando, FL; special issue of *SIGPLAN Notices,* Vol 22, No 12, Dec 1987, pp 168-182.

[WEK90] A.L. Winblad, S.D. Edwards, and D.R. King. *Object-Oriented Software.* Addison-Wesley Publishing Co, Reading, Mass, 1990.

[WW89] R. Wirfs-Brock and B. Wilkerson. "Object-Oriented Design: A Responsibility Driven Approach", *OOPSLA'89 Proceedings,* Oct 1989, New Orleans, LA; special issue of *SIGPLAN Notices,* Vol 24, No 10, Oct 1989, pp 71-75.

[WWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, Inc, Englewood Cliffs, NJ, 1990.

DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2 copies
Center for Naval Analyses 4401 Ford Avenue Alexandria, VA 22302-0268	1 copy
Director of Research Administration Code 81 Naval Postgraduate School Monterey, CA 93943	1 copy
Maj M.L. Nelson, USAF Naval Postgraduate School Code CS, Dept. of Computer Science Monterey, CA 93943	5 copies
Professor LuQi Naval Postgraduate School Code CS, Dept. of Computer Science Monterey, CA 93943	5 copies
Capt E.G. de Paula, Brazilian AF Centro Técnico Aeroespacial (CTA) IAE - ESB São José dos Campos, SP, Brazil, 12225	5 copies